VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Modern C++ for Data Structures and Algorithms

Duomenų Struktūros ir Algoritmai Moderniame C++

Course Project

Author:	4 kurso I grupės studentas
	Domas Kalinauskas
Supervisor:	Viktoras Golubevas

Vilnius – 2025

Contents

Introduction

Intro

The programming landscape is rapidly evolving, with an increasing focus on creating safer software. Memory safety issues, often a source of critical vulnerabilities, have led to the widespread adoption of languages such as Rust, which enforces safety guarantees at compile time, and managed, garbage-collected runtimes like Java and .NET. C++ remains a cornerstone of systems programming, however, this comes with challenges – traditionally developers must navigate a steep learning curve and manually manage safety concerns such as resource allocation and deallocation, alongside avoiding and mitigating undefined behavior. To address these issues, modern iterations of C++ have introduced features aimed at improving safety, usability, and expressiveness, making the language more competitive and accessible.

This course project aims to analyze these advancements in C++ to better understand their real-world impact on both safety and performance. Additionally, it is important to evaluate how these features compare to similar capabilities in other languages, in order to understand where C++ might fall behind and could improve in the future.

Objectives

1. Evaluate the usability and performance of ranges
2. Compare SFINAE with concepts
3. Analyze type-safe data structure additions in STL
4. Detail constant expression function capabilities
5. Introduce coroutines

1 Modern C++ Features for Algorithmic Problem Solving

1.1 Ranges

1.1.1 Index-based iteration

A common pattern in C++ code is applying an operation over a selection of elements. The most classic style, which is still in-use today is the index based loop seen in fig. 1

```cpp
void operate(std::span<uint32_t> values) {
    for (size_t i = 0; i < values.size(); ++i) {
        std::print("Value: {}", values[i]);
    }
}
```

Figure 1. index-based loop

There are a few notable downsides to the index based for loop, namely that it's error-prone, and can only be effectively used with random-indexable types (e.g. std::array, std::vector). This means that if we want to iterate over a list, or another custom container, we'd have to change the for loop structure to be compatible.

1.1.2 Value-based iteration

With the introduction of C++11, we got access to the iterators and the range-based for loop[1] seen in fig. 2

```cpp
void operate(const auto &container) {
    for (const auto& x : container) {
        std::print("Value: {}", x);
    }
}
```

Figure 2. range-based loop

When using iterators with range-based for loops, we no longer have to manually write the iteration code, meaning it doesn't matter if the type is a array, vector, list, or any other custom iterator. Under the hood, these range-based for loops relies on the container having .begin()/.cbegin() and .end()/.cend() member functions – they return corresponding iterators which allow access to values.

These iterators form the basis of the standard library algorithms and ranges. They can be thought of as the "glue" between data structures and algorithms. Iterators provide a generic way to navigate through the elements in a sequence. This approach allows separating the algorithm from the container and their internal data layouts, thus making code more flexible [AS20]. However,

[1]for production you'd want to use a specific concept instead of auto – that way you would get a clearer error message

iterators can become quite hard to follow or reason about when combining some of the more complex standard library operations.

1.1.3 Ranges improvement over iterators

Take for example, we had a scenario where we're given a collection of numbers, we want to skip the first N values, filter out the even ones, then apply some operation on them, and finally print them out - all while quitting early, if we encounter some magic number. The range-based filter loop approach can be seen in fig. 3

```
static double transform_number(int input);
static void print(double val);
void operate(const auto &container, size_t offset, double early_exit) {
    /* Skip first 3 elements (container.begin() + offset isn't supported for all iterator
    std::find_if(std::next(container.begin(), offset), container.end(), [&](int val) {
        /* Filter only for odd values*/
        if (!is_odd(val)) {
            return false;
        }

        /* Apply operation */
        double transformed = transform_number(val);

        /* Check if early exit encountered */
        if (transformed == early_exit) {
            return true;
        }

        /* Else - print value & continue */
        print(transformed);
        return false;
    });
}
```

Figure 3. range-based filter loop

This approach is quite verbose - We have to use std::next to ensure support for (most) iterator types, pass the begin and end manually, use std::find_if (keeping in mind that false means to continue iterating - normally you'd expect the inverse). While it isn't impossible to understand, but it's certainly not clear at a glance what the code is doing (or why it's doing it), especially when someone might not know exactly what is going on.

This becomes much simpler when we use ranges, introduced first in C++20. With ranges and views[2], the same functionality can be implemented in a much simpler way, as seen in fig. 4

[2]special type of range, where the operations are lazy

```cpp
static double transform_number(int input);
static void print(double val);
void operate(const auto &container, size_t offset, double early_exit) {
    auto elems = container
        | std::views::drop(offset)
        | std::views::filter(is_odd)
        | std::views::transform(transform_number)
        | std::views::take_while([&](auto v){return v != early_exit;});
    std::ranges::for_each(elems,print);
}
```

Figure 4. ranges filter view

1.1.4 Ranges pitfalls

However, C++ ranges, and in particular, lazy views, suffer from some less than obvious safety issues and performance downsides. Such as the filter view being incompatible with modifications that change values in a way that a previously matching entry leads to it no longer satisfying the filter. Doing such an action is considered undefined behaviour by the standard, which means that the pattern of using a filter to *fix* some attribute of elements cannot safely be done [Jos23].

Going back to our view filter example, after modifying it to print what action is being executed, the result with some sample data can be seen in fig. 5.

The unexpected part here is that we have to apply transformation twice for each value passing filter - once when evaluating whether value is taken, and then again when we dereference it. This is because, semantically, in the C++ iterator model, positioning (++) and accessing (*) are distinct operations. Under the hood, take must access (*) the value, then change positioning (++) of the iterator. After the value is taken, then our for_each does the same thing - it dereferences whatever we get out, meaning that the transformation gets applied twice, since there's no way of 'reusing' the accessed value.

```cpp
static double
transform_number(int input) {
    return input * 5;
}

int main() {
    static constexpr
    auto nums = std::array{
        1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12
    };
    operate_ranges(nums, 3, 45.0);
    return 0;
}
```

```
drop
drop
drop
/* This drop 'opens' the flood gates */
drop
/* Filter out even value */
filter
/* Value passes filter*/
filter
/* Value is transformed */
transform
/* We execute take */
take
/* Value is taken and then... transformed? */
transform
/* Value is printed */
25.000000
/* ... continue as above */
filter
filter
transform
take
transform
35.000000
filter
filter
transform
/* Value doesn't pass take
- early exit condition */
take
```

Figure 5. C++ range code and output

1.1.5 Comparison with Rust iterators

This differs from, for example, the Rust iterator model, where essentially each stage of the pipeline passes values directly to the next. This becomes apparent when you want to implement your own Rust iterators vs C++ views. In the C++ case, the view takes some input view, and then must define both a begin() and an end() operation which return 'iterators', whereas with the Rust variant, only a next() function, returning an Optional<ValueT> is required.

That same example given in fig. 5 but written using Rust iterators, along with the produced output can be seen in fig. 6

```rust
fn operate_ranges<T>(
    container: T,
    offset: usize,
    early_exit: f64,
) where
    T: IntoIterator<Item = i32>,
{
    container.into_iter()
        .skip(offset)
        .filter(|&v| { is_odd(v) })
        .map(|v| { transform_number(v) })
        .take_while(|&v| { v != early_exit})
        .for_each(|v| print(v));
}
```

```
drop
drop
drop
drop
filter
filter
transform
take
/* Up to here, matches C++. We see
here, that when using the Rust model,
the transformation doesn't need
to be applied again */
25
filter
filter
transform
take
35
filter
filter
transform
take
```

Figure 6. Rust iterator code and output

1.1.6 Avoiding duplicate work

Duplicate work can be avoided in the C++ case, by using a view that caches the result of dereferencing an element, but the downside is that, depending on what it is you're caching, it can become quite expensive. Unfortunately, while such a view was present in ranges-v3 [3] - P0896R4[4], the proposal that got accepted into the C++20 standard, had quite a few of the views in ranges-v3 missing, views::cache1 being one of them.

[3]code basis of Ranges-TS proposal

[4]`https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf`

1.1.7 Ranges benchmark

To verify whether there are any glaring performance differences these approaches, a micro-benchmark was performed[5]. The benchmark consisted of four test cases:

- regular for loop
- find_if
- view
- view, but using anonymous lambdas

The results of this benchmark can be seen in fig. 7. The two interesting parts are that find_if **outperforms** the regular loop approach, and that the view with lambdas is more performant than the view without. After performing an analysis on the generated assembly, the author concludes that on the tested compiler, find_if was auto-vectorized to perform 8 operations per each loop, meaning that N + K 'iterations' were performed, instead of 8 x N + K if there was no vectorization. Why find_if was vectorized whereas the raw loop wasn't - the author wasn't able to determine for sure, however it could be due to the starting offset from the for loop preventing that optimization. The other interesting part - view with lambdas outperforming the view without is equally as puzzling. Checking the assembly, when range functions were provided plain anonymous lambdas that take the input and pass it straight to the wanted function, gcc decides to inline the wanted function body inside the anonymous lambda (and then inline the lambda within the corresponding range code), whereas when passed straight function pointers, gcc opted to call the function within the range code instead - without performing inlining. The author also double-checked generated assembly with gcc trunk and clang trunk. Clang trunk recognizes and inlines the function, whereas gcc trunk does the same as the gcc used to perform the benchmark.

[5] note - performance can vary depending on compiler and compiler version, for details see appendix

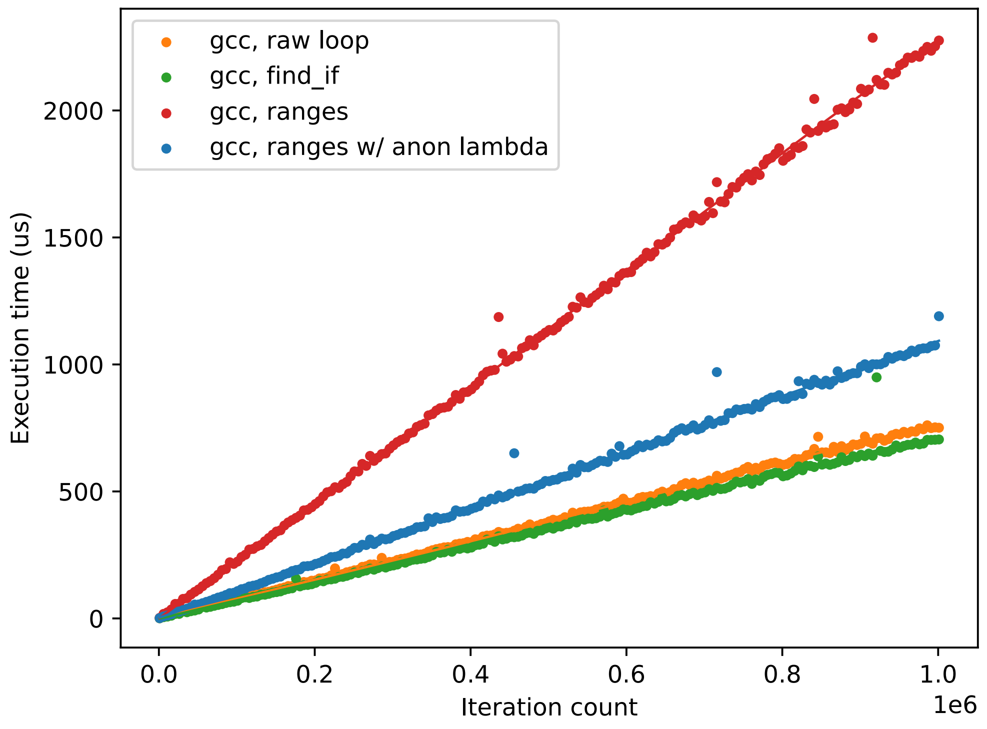

Figure 7. Ranges performance comparison

1.1.8 Ranges Summary

In the authors opinion, ranges increase code readability with minimal performance downside and should be preferred, except in areas that show poor performance and where that extra performance boost is needed (verified with a benchmark and/or profiling).

1.2 Concepts

1.2.1 SFINAE introduction

When writing generic code, it's good practice to specify the requirements that the algorithm or function has.

Previously, the best way of doing that, would be to have a static_assert inside the implementation when using a function, in combination with SFINAE[6] structs to check for the appropriate member functions, as seen in fig. 8

However, there are a few problems with this approach. SFINAE checks are quite hard to understand and write, if you haven't used them before. Alongside that, the requirement itself is not mentioned anywhere in the function signature - so the only way to know is either by reading a comment (if there is one), inspecting the source code, or trying to compile and seeing what specifically fails. When compiling an example that doesn't pass the static assert, you'll get not only

[6]SFINAE - Substitution failure is not an error

```cpp
struct Point {
    int x, y;
};

struct Square {
    size_t size;

    constexpr size_t area() const noexcept {
        return size * 2;
    }
};

template <typename T, typename = void>
struct HasArea : std::false_type {};

template <typename T>
struct HasArea<T, typename std::enable_if<
    std::is_member_function_pointer<decltype(&T::area)>::value>::type
> : std::true_type {};

template <typename T, typename F>
static constexpr int order_areas(const T& a, const F& b) {
    static_assert(HasArea<T>::value, "arg a doesn't have area()");
    static_assert(HasArea<F>::value, "arg b doesn't have area()");
    size_t a_area = a.area();
    size_t b_area = b.area();
    if (a_area == b_area) {
        return 0;
    } else {
        return a_area > b_area ? 1 : -1;
    }
}
static_assert(order_areas(Point{0, 0}, Square{4}) == 0);
```

Figure 8. SFINAE struct - area member function

the failed static_assert message, but also all areas that don't conform to that requirement. In this example where we only use area(), it might help pinpoint what's exactly wrong – but when more complex type requirements are in play, the multiple error messages hurt more than they help.

1.2.2 Concepts advantages

C++20 concepts improve on this, by removing the need to write SFINAE structs while also allowing us to specify the concept that our type must match inside the function signature, as seen in fig. 9. The result is that error messages become much more clear, and the requirements of a function are visible from it's declaration. Essentially, concepts allow developers to define a set of predicates that a type must satisfy, offering a more structure and readable way to constrain templates, as opposed to the SFINAE approach [Far24].

```
template <typename T>
concept HasArea = requires(T t) {
    { t.area() } -> std::same_as<std::size_t>;
};

static constexpr int order_areas(const HasArea auto& a, const HasArea auto& b) {
    size_t a_area = a.area();
    size_t b_area = b.area();
    if (a_area == b_area) {
        return 0;
    } else {
        return a_area > b_area ? 1 : -1;
    }
}
static_assert(order_areas(Point{0, 0}, Square{4}) == 0);
```

Figure 9. Concept – area member function

1.2.3 Constructor and Destructor requirements

Concepts allow specifying requirements for constructors and destructors. One example use case would be with trivial destructors and an optional type. Plainly, a trivial destructor is one that *doesn't* have to be executed. Types which don't have a user-defined destructor and all of their member variables are trivially destructable – are also considered trivially destructable. For an optional this attribute of a provided type could be used to omit a check of existence and a call to the empty destructor. To ensure that optionals with trivially destructable members are also trivially destructable, we have to have two destructors – one that checks existence + calls destructor, and one that does nothing (= default). Before concepts, such an implementation would require a dispatch to one of two base classes – which both have a common base class that implements the functionality. After the appearance of concepts, a much simpler approach can be taken, as seen from the side-by-side in fig. 10

```cpp
template <typename T>
struct optional_base {
    /* <impl details> */
};

template <typename T>
struct optional_trivial
: public optional_base<T>{
    /* if trivial destr - do nothing */
    ~optional_trivial() = default;
};

template <typename T>
struct optional_nontrivial
: public optional_base<T> {
    /* else only call destr if activated */
    ~optional_nontrivial() {
        if (this->has_value()) {
            this->value.~T();
        }
    }
};

template <typename T>
using optional = std::conditional_t<
    std::is_trivially_destructible_v<T>,
    optional_trivial<T>,
    optional_nontrivial<T>
>;
```

```cpp
template <typename T>
struct optional {
    /* <impl details>*/

    /* if trivial destr - do nothing */
    ~optional() requires
        std::is_trivially_destructible_v<T>
            = default;

    /* else only call destr if activated */
    ~optional() {
        if (has_value()) {
            value.~T();
        }
    }
};
```

Figure 10. Optional SFINAE vs concepts

1.2.4 Concept compilation speed benchmark

Furthermore, concepts can improve compile times, since for SFINAE the compiler must instanciate all variants before picking one[7] from the list of available variants[8], whereas concepts allow short-circuiting, meaning that as soon as one part of concept doesn't match – evaluation is finished. To confirm that this is indeed the case, a compile-time test was performed. A recursive base class with templated value N was created, which contains itself with N-1 as it's base class all the way to zero. Then, a function specialization for each specialization of that base class from N to zero was added. This essentially tests how fast the compiler can instantiate and pick the correct overload, leveraging SFINAE template with std::enable_if_t from STL, and the same equivalent functionality with the concept keyword *requires*. For the SFINAE implementation, the compiler must instantiate and parse all implementations even if the passed recursive base class doesn't match or won't be able to match the overload set. This is confirmed by the performance graph in fig. 11. It shows that concepts are slightly faster to compile than their equivalent SFINAE-based implementations, since they don't have to instantiate all function specializations and base classes.

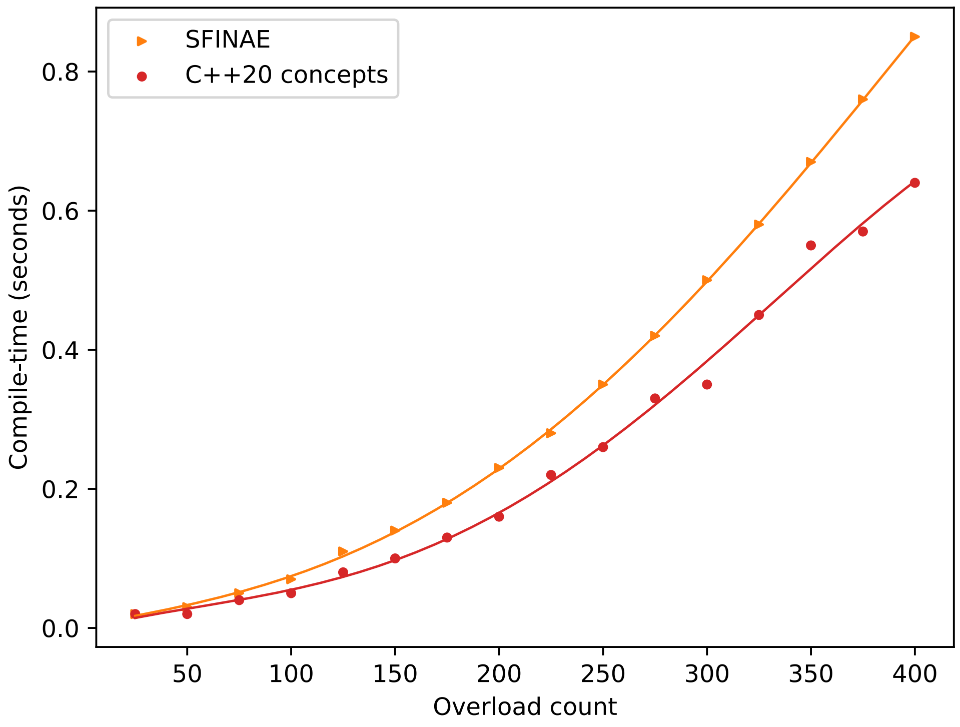

Figure 11. Concept vs SFINAE compilation time

[7]the most 'specific' one is preferred

[8]if >1 applicable variant of identical specificity, compile-erorr about ambigiuty is emitted instead.

1.2.5 Comparison with Rust traits

For comparison in Rust, traits are used both for monomorphised functions (templates / concepts), but also for dynamic dispatch (interfaces / virtual functions). While C++ concepts apply to any type which satisfies it's constraint, Rust traits only apply when *explicitly* implementing that trait for a type. This prevents cases where an identical but semantically different function matches[9]. The same example but implemented via Rust traits can be seen in fig. 12

```rust
struct Point {
    pub x: u32,
    pub y: u32
}

struct Square {
    pub size: u32
}

trait HasArea {
    fn area(&self) -> u32;
}

impl HasArea for Square {
    fn area(&self) -> u32 {
        self.size * self.size
    }
}

fn order_areas(a: &impl HasArea, b: &impl HasArea) -> i32 {
    let a_area = a.area();
    let b_area = b.area();
    if a_area == b_area {
        0
    } else if a_area > b_area {
        1
    } else {
        -1
    }
}

fn try_order(a: &Point, b: &Square) -> i32 {
    /* Won't compile - Point
    doesn't have `HasArea`
    trait implemented*/
    order_areas(a, b)
}
```

Figure 12. Rust trait - area function

[9] e.g. draw() - could apply to drawing on a canvas, or drawing a card

1.3 Type-Safe Data Structures

Recent releases of the STL and C++ standard have contained more type-safe additions, which simplify the way of *correctly* handling combination types, such as unions or optional values. They also provide a way of handling, storing and dispatching against all possible types for a function without having to resort to virtual functions or memory allocations.

1.3.1 Type-safe Union

The older approach to having N variants in a single-type in C and C++ was having a union, however with the appearance of constructors/destructors, when combining them with unions manually, it is very trivial to introduce an invalid memory access or something similiar. This approach requires a bit of boilerplate - adding an enum indicating which value it has, then manually handling the constructor and destructor logic to ensure that the correct object gets initialized/deinitialized, as seen in fig. 13

```
struct Variant {
    enum variant_type {
        numeric,
        string,
        multi_data
    };
    variant_type type;
    union d {
        uint32_t numeric
        std::string string;
        std::array<uint8_t, 2> multi_data;
    };

    Variant() {
        /* Manual constructor depending on which type*/
    }
    ~Variant() {
        /* Manual destructor depending on which type */
    }
};
```

Figure 13. C++ union

To handle this, a type-safe union was introduced in C++17 - std::variant. It is essentially a "smart union" in that in addition to storing a value of one of the specified types at a time, the variant knowns which type it contains, while with a union, the programmer is responsible for reading the same type as was written. The variant also handles calling the destructor that corresponds to the currently stored type when the object goes out of scope, or when switching the contained type. Accessing the wrong type throws an exception, though a non-throwing variation is possible by checking whether the contained type matches the expected one before accessing [Pik23]. When comparing it to Rust - it lags behind the Rust languages equivalent, since we can't indicate the slot without creating a separate enum, meaning that access is done either via concrete index (0, 1, etc.), or, if there is only a single instance of concrete type, then via type (uint32_t, std::string, etc.). Furthermore, the logic required to match is much cleaner in Rust, with the C++ style std::visit being much more complex. An example of variant and visiting can be seen in fig. 14

```cpp
using DataType =
    std::variant<uint32_t, std::array<uint8_t, 2>, std::string>;

template<class... Ts>
struct overloaded
    : Ts... { using Ts::operator()...; };

void dispatch(const DataType &data) {
    std::visit(overloaded{
        [](uint32_t num)
            { std::print("Numeric: {}\n", num); },
        [](const std::array<uint8_t, 2> &arr)
            { std::print("Multi: {} {}\n", arr[0], arr[1]); },
        [](const std::string &str)
            { std::print("String: {}\n", str); }
    }, data);
}
```

Figure 14. C++ variant

Rust enums are unique in that they provide the ability to have different data types for each of the enum variants, which can then safely be dispatched, as can be seen in fig. 15

```rust
enum DataType {
    Numeric(u32),
    MultiData(u8, u8),
    Text(String),
}

fn dispatch(data: DataType) {
    use crate::DataType::*;
    match data {
        Numeric(num)
            => println!("Numeric: {}", num),
        MultiData(d1, d2)
            => println!("Multi: {},{}", d1, d2),
        Text(string)
            => println!("String: {}", string)
    }
}
```

Figure 15. Rust enum

1.3.2 Optional value

Another quite common pattern encountered is the ability to indicate the absence of a value.

In Rust, this is achieved via Option<T>, which is an enum variant with 2 possible values - None, and Some(T) (where T is the type of the optional). Due to this, the Rust optional can re-use alot of the matching syntax that rust makes available. C++ also has a similiar feature available in the standard library - std::optional<T>. An example of both can be seen in fig. 16

```cpp
static std::optional<int>
perform_calculation(int input) {
    if (input % 2 != 0) {
        return std::nullopt;
    }
    return input * 3;
}

static void
use_optional(int input) {
    auto res = perform_calculation(input);
    if (res) {
        std::cout << *res;
    } else {
        std::cout << "error";
    }
}
```

```rust
fn perform_calculation
        (input: i32)-> Option<i32> {
    if input % 2 != 0 {
        None
    } else {
        Some(input * 3)
    }
}

fn use_optional(input: i32) {
    match perform_calculation(input) {
        Some(res) => println!("{}", res),
        None => println!("error"),
    }
}
```

Figure 16. C++ and Rust optional comparison

1.3.3 Result value

Sometimes functions need to return either a success or failure value, which are of different types. For example, either a calculated value or a string error message. In C++ this is handled with a std::expected<T, F>, where T is the success type, and F is the failure type. The equivalent in Rust, is Result<T, F>. This, just like Option, relies on enum variants, which allows it to re-use the matching logic. Example of C++ and Rust seen in fig. 17

Essentially, the Rust takes one feature (enum variants), and then applies it to their 'functional' types. This means that optimizing one part, e.g. enum variant matching, will materialize as a gain in both Optional and Expected. This is in contrast to C++, where std::variant, std::optional and std::expected are all distinct types with their own implementation details.

```cpp
static std::expected<int, std::string_view>
perform_calculation(int input) {
    if (input % 2 != 0) {
        return std::unexpected{
            "Input must be odd"
        };
    }
    return input * 3;
}

static void
use_expected(int input) {
    auto res = perform_calculation(input);
    if (res) {
        std::print("{}", *res);
    } else {
        std::print("Err: {}",
            res.error());
    }
}
```

```rust
fn perform_calculation
        (input: i32)
        -> Result<i32, &'static str> {
    if input % 2 != 0 {
        Err("Input must be odd")
    } else {
        Ok(input * 3)
    }
}

fn use_result(input: i32) {
    match perform_calculation(input) {
        Ok(res)
            => println!("{}", res),
        Err(msg)
            => println!("Err: {}", msg),
    }
}
```

Figure 17. C++ and Rust std::expected/Result comparison

1.3.4 Discriminant size optimization

When using Rust, the compiler can optimize the enum instances' discriminant[10] to use invalid byte patterns instead of a separate field [Hus24]. This can lead to significant final type size savings, esp. when even a 1-byte discriminant would require aligning to the requirements of the biggest type. So e.g. for a type with 8 byte alignment, if no invalid discriminant patterns exist, the final size would be 16 bytes.

A concrete example of this in use, is with std::optional<bool> - this takes up 2 bytes, whereas in Rust, Option<bool>, would only take up 1 byte of space, since the unused bits of the bool value can be used to store the discriminant information. Assertions that this is the case can be seen in fig. 18

```cpp
/* Compilation will fail if
size isn't equal to 2 */
static_assert(
    sizeof(std::optional<bool>) == 2
);
```

```rust
/* Compilation will fail if
size isn't equal to 1 */
const SizeChecker: [u8; 1]
= [0; std::mem::size_of::<Option<bool>>()];
```

Figure 18. C++ and Rust optional bool size

In the absence of specializations for C++, there are third-party solutions[11] for C++ that take advantage of these unused bits, however, having something that's built into the language would be much more convenient.

[10]element that indicates which variant is being held

[11]size-optimized std::optional - https://github.com/Sedeniono/tiny-optional

2 Modern C++ for High-Performance Computing

2.1 Compile-time calculations

2.1.1 Intro to constexpr

C++ has the ability to move calculations to build-time - most commonly used with constexpr, consteval, constinit keywords.

In certain scenarios, constexpr calculations can provide a performance boost by moving work from runtime execution into compile-time. It's important to note, that *constexpr doesn't guarantee compile-time execution*, only allows it. This is in contrast to the consteval specifier - this forces compile-time evaluation - meaning that they can't even be used in a runtime context. Similarly, constinit cannot be used in a run-time context since it is used to guarantee that a variable is initilized at compile-time.

2.1.2 Avoiding Static Initialization Order Fiasco

Constinit is used to solve a problem that appears when dealing with multiple global static variables - the "Static Initialization Order Fiasco". If two static objects, x and y exist in separate translation units, and y depends on the initialization of x, there are no guarantees that when y constructor is called, that x will have been already constructed. A common workaround is to use the "Construct On First Use" idiom - the idiom describes a function that returns a reference (or pointer) to a locally contained static variable - this way the initialization of that static is performed only when that function is called. However, the easiest way to ensure that this can't happen is to guarantee constant time initialization - by having the constinit specifier on any static variables.

2.1.3 Constexpr use-cases in practice

There are many potential use cases, for example:

- Offloading math calculations.
- Pre-compiling Regex expressions[12].
- When all keys (or even values) of a map / dictionary are known at compile time, then we can generate optimal hashes / layouts, as well as removing the need for dynamic allocation. Avoiding allocations were possible is useful in both HPC contexts, but also in the authors field of work - embedded devices[13].
- Embedding data inside executable, while also compressing it [14]

As newer versions of the C++ standard were released - various improvements and requirement relaxations for constexpr calculations were introduced, ranging from allowing dynamic allocations

[12]`https://github.com/cmargiotta/e-regex`

[13]`https://github.com/lynxcs/heurohash`

[14]`https://github.com/AshleyRoll/squeeze`

in constexpr (provided they don't outlive the constexpr context), to adding constexpr compatibility to large swaths of the standard library.

2.1.4 Constexpr limitations

There's a small downside to constexpr though – all the information must be visible to the compiler in the same translation unit, meaning that constexpr functions must be either header-only, or that the constant folding can only be performed inside of the translation unit that contains that constexpr function. Hiding dependencies or implementation details becomes functionally impossible in the header-only case. Also, for every translation unit that uses constexpr data the instantiation must be done separately, meaning that expensive to calculate constexpr functions which are included in many source files can significantly slow down compile-times.

The multi-calculation issue is partly solved by C++ modules, however as of writing, first-class support for modules is still quite lacking in all 3 major compilers (gcc, clang, MSVC).

2.1.5 Comparison with Rust equivalent

Rust has 2 features which compete with constexpr:

- const functions
- procedural macros

Const functions in Rust are much more limited than their C++ equivalent – they can't allocate and Traits can't have them. For simple jobs, such as having some data type layout defined as a constant, or performing trivial operations, Rust const functions are suitable. An example of a constexpr C++ function and it's const fn Rust equivalent can be seen in fig. 19

Procedural macros on the other hand are much more powerful, since they are evaluated during AST construction – before IR generation. Essentially, they allow running code at compile time that can transform the AST – they have access to the same resources that the compiler has. They can read and execute arbitrary code, which is a step above of what's currently possible in C++.

One commonly used Rust feature, is #[derive(...)]. This feature leverages macros to automatically generate code, for example, to print the contents of the struct, or provide copy, clone, etc. functionality.

However, procedural macros come with a downside in that **creating** simple functions with them is much more complex than in their constexpr C++ counterpart.

In the near future C++26 *should* receive very similiar functionality (though not quite reaching what's possible with Rust macros) in the form of reflection. The key proposal in this area is P2996 which "is a proposal for a reduced initial set of features to support static reflection in C++" [CDK+24] alongside a few follow-up / modification proposals, such as P3294 which suggests modifying P2996 "to add code injection in the form of token sequences" [ARV24].

```cpp
/* Can be called in
run-time or compile-time */
static constexpr
int multiply(int a, int b) {
    return a * b;
}

/* Guarantee compile-time execution*/
static constexpr
    auto mult = multiply(1, 2);

int runtime_func(int a, int b) {
    /* Can also be
    used at run-time*/
    return multiply(a, b);
}

/* Can only be called
at compile-time */
static consteval
int multiply_c(int a, int b) {
    return a * b;
}

int runtime_func_c(int a, int b) {
    /* Will not compile, consteval
    function cannot be called at
    run-time*/
    return multiply_c(a, b);
}
```

```rust
/* Can be called in
run-time or compile-time */
const fn
multiply(a: i32, b: i32) -> i32 {
    return a * b;
}

/* Guarantee compile-time
execution */
const mult: i32
    = multiply(1, 2);

pub fn
runtime_func(a: i32, b: i32) -> i32 {
    multiply(a, b);
}
```

Figure 19. C++ and Rust constant functions

2.2 Coroutines

2.2.1 Coroutine theory

Starting with C++20, the language has access to coroutines. Coroutines are a generalisation of a function that allows the function to be suspended and then later resumed. A typical function usually has 2 operations - call and return. Coroutines expand that with 3 additional parts - suspend, resume, destroy. Suspending allows the coroutine to transfer execution back to the caller / resumer of the coroutine. Resuming continues execution of a previously suspended coroutine. Destroy cleans up all values that were currently in scope [Bak17]. From an implementation point of view, coroutines can be either stackful or stackless. Stack in this case refers to where the coroutine frame is stored. In a stackful coroutine, the frame data is stored on the stack, whereas stackless stores the coroutine data separately from the callers stack. C++ coroutines adopt the stackless approach.

2.2.2 Coroutine frame allocation

It's worth noting, that one of the main parts of coroutines, the coroutine frame, is typically allocated on the heap. The standard does allow for compilers to implement a HALO (Heap allocation ellision optimization) under some conditions, however, there is no way to guarantee it [SN18]. This means that coroutines cannot or should not be used in some contexts, such as safety critical applications or those that require deterministic execution times.

2.2.3 Coroutine requirements

A function is considered a coroutine if it uses any of the following keywords:

- co_await - suspends execution until resumed
- co_return - complete execution, optionally return a value
- co_yield - suspend execution and return a value

However, not all functions can be coroutines. The standard currently prevents constructors, destructors, constexpr functions, vararg functions, functions that return auto or a concept type, and main() from being coroutines [Ban24].

2.2.4 Coroutine switching latency

The switching latency between coroutines (meaning the suspend + resume latency), is quite small, and depending on the CPU, can take < 1ns. This can be leveraged to improve CPU utilization, via interleaving of multiple coroutine frames. Using an interleaved binary search results in almost 3.5x speed improvement over traditional naive 1-by-1 binary search. This is achieved by hinting to the CPUs MMU to prefetch data before performing the search [Nis18].

2.2.5 std::generator comparison with Python generators

When first introduced, C++ coroutines were bare - the STL provides almost no additional functionality built on top of the language feature, meaning that all coroutine supporting code needed to be written from scratch.

Starting with C++23, the STL included std::generator, which allows generating a sequence of elements. The behaviour is very similiar to Pythons generators. Implementations of a endless fibonnaci sequence generator written in python and C++ can be seen in fig. 20

```python
def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

def fib_seq(max_cnt):
    f = fibonacci()
    for i in range(max_cnt):
        print(next(f))
```

```cpp
std::generator<uint64_t> fibonacci() noexcept {
    size_t a = 1, b = 1;
    while (true) {
        co_yield a;
        a = std::exchange(b, a + b);
    }
}

void fib_seq(size_t max_cnt) noexcept {
    auto fib = fibonacci()
            | std::views::take(max_cnt);
    for (auto val : fib) {
        std::cout << val << '\n';
    }
}
```

Figure 20. Python and C++ fibonnaci generator

Results and conclusions

The findings of this project highlight both the strengths and shortcomings of modern C++. The benchmarks performed using Ranges show that they significantly enhance code clarity while delivering performance on par with traditional approaches. However, certain edge cases, such as potentially unsafe use of filters and the redundant repetition of work in certain views, highlight a few areas where improvements are needed. Concepts offer a much cleaner way to define and enforce type requirements while also making compilation errors more clear for types that don't meet them. They also enhance compile-times when compared to SFINAE, but they still trail behind similar features in other languages like Rust in terms of flexibility and usability. Type-safe additions to the STL simplify resource management and reduce errors by handling RAII semantics automatically, though they sometimes lack the expressiveness of counterparts in other programming languages, such as the match syntax in Rust. Finally, coroutines enable the simplification of asynchronous workflows by allowing users to express code in a more linear way, while removing the need for callback-based generator functions, and exhibiting a very low switching latency. Yet limitations like heap allocation overhead and restrictions on coroutine usage in certain contexts showcase areas where they could use some further work.

While overall, the language is not perfect, C++ remains widely used and in active development. It has a steady influx of proposals aimed at addressing the languages limitations. Constant evolution in the ecosystem ensures that C++ continues to be competitive in this rapidly changing landscape.

1. Ranges improve readability and extensibility, with minimal performance losses
2. Concepts make compilation-errors more readable while decreasing compile-times
3. STL contains most commonly found type-safe functional style data structures, which increases safety and convenience over manual implementations.
4. Coroutines can be used to turn callback based code into linear-looking code, while minimizing latency cost

References

[ARV24] N. A. Alexandrescu, B. Revzin, D. Vandevoorde. *Code Injection with Token Sequences*. 2024. [visited on 2024-12-21]. Available from: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3294r2.html`.

[AS20] B. Andrist, V. Sehr. *C++ High Performance, 2nd edition*. Packt Publishing, 2020. 140–145 pages.

[Bak17] L. Baker. *Coroutine Theory*. 2017. [visited on 2024-12-22]. Available from: `https://lewissbaker.github.io/2017/09/25/coroutine-theory`.

[Ban24] M. Bancila. *Modern C++ Programming Cookbook, 3rd edition*. Packt Publishing, 2024. 758–766 pages.

[CDK+24] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, D. Vandevoorde. *Reflection for C++26*. 2024. [visited on 2024-12-21]. Available from: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2996r8.html`.

[Far24] J. Farrier. *Data Structures and Algorithms with the C++ STL*. Packt Publishing, 2024. 390–400 pages.

[Hus24] E. Huss. *The Rust Reference*. 2024. [visited on 2024-12-22]. Available from: `https://doc.rust-lang.org/stable/reference/items/enumerations.html`.

[Jos23] N. Josuttis. *C++ Standard Views*. 2023. Available also from: `https://www.youtube.com/watch?v=qv29fo9sUjY`. ACCU 2023.

[Nis18] G. Nishanov. *Memory Latency Troubles You? Nano-coroutines to the Rescue! (Using Coroutines TS, of Course)*. 2018. Available also from: `https://www.youtube.com/watch?v=j9tlJAqMV7U`. CppCon 2018.

[Pik23] F. G. Pikus. *Hands-On Design Patterns with C++, 2nd edition*. Packt Publishing, 2023. 550–556 pages.

[SN18] R. Smith, G. Nishanov. *Halo: coroutine Heap Allocation eLision Optimization: the joint response*. 2018. [visited on 2025-01-15]. Available from: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html`.

Abbreviations

1. SFINAE – Substitution Failure Is Not An Error
2. Constexpr – Constant Expression
3. STL – Standard Template Library
4. RAII – Resource Acquisition Is Initialization
5. MMU – Memory Management Unit
6. AST – Abstract Syntax Tree
7. IR – Intermediate Representation

Appendixes

Appendix 1

Benchmark details

All source code for benchmarks can be found in this course projects public repository: `https://github.com/lynxcs/vu_kursinis`

The benchmarks were performed on machine - Linux 6.12, AMD 5900x, 32GB RAM.

Used compilers in project were:

- gcc 12.2 - For running benchmarks
- gcc trunk - For verification whether assembly improved - based on 15.0 at time of testing (2024-12-30)
- clang trunk - For verification whether assembly improved - based on 20.0 at time of testing (2024-12-30)